Precise Runahead Execution

Ajeya Naithani* Josué Feliu†‡ Almutaz Adileh* Lieven Eeckhout*
*Ghent University, Belgium †Universitat Politècnica de València, Spain

***Abstract*—Runahead execution improves processor performance by accurately prefetching long-latency memory accesses. When a long-latency load causes the instruction window to fill up and halt the pipeline, the processor enters runahead mode and keeps speculatively executing code to trigger accurate prefetches. A recent improvement tracks the chain of instructions that leads to the long-latency load, stores it in a runahead buffer, and executes only this chain during runahead execution, with the purpose of generating more prefetch requests. Unfortunately, all prior runahead proposals have shortcomings that limit performance and energy efficiency because they release processor state when entering runahead mode and then need to re-fill the pipeline to restart normal operation. Moreover, runahead buffer limits prefetch coverage by tracking only a single chain of instructions that leads to the same long-latency load.**

We propose *precise runahead execution (PRE)* which builds on the key observation that when entering runahead mode, the processor has enough issue queue and physical register file resources to speculatively execute instructions. This mitigates the need to release and re-fill processor state in the ROB, issue queue, and physical register file. In addition, PRE pre-executes only those instructions in runahead mode that lead to full-window stalls, using a novel register renaming mechanism to quickly free physical registers in runahead mode, further improving efficiency and effectiveness. Finally, PRE optionally buffers decoded runahead micro-ops in the front-end to save energy. Our experimental evaluation using a set of memory-intensive applications shows that PRE achieves an additional 18.2% performance improvement over the recent runahead proposals while at the same time reducing energy consumption by 6.8%.

I. Introduction

Runahead execution [18, 40, 42] improves processor performance by accurately prefetching long-latency loads. The processor triggers runahead execution when a long-latency load causes the instruction window to fill up and halt the pipeline. Instead of stalling, the processor removes the blocking long-latency load and speculatively executes subsequent instructions to uncover future independent long-latency loads and expose memory-level parallelism (MLP). The processor terminates runahead execution and resumes normal operation when the stalling load returns. Because runahead execution generates memory loads by looking at the application's code ahead of time, the prefetch requests it generates are accurate, leading to significant performance benefits.

Unfortunately, the performance benefits of runahead execution are limited by its prefetch coverage and the overheads associated with speculative code execution. Prefetch coverage relates to the number of *useful* prefetch requests generated in runahead mode. The higher the prefetch coverage in runahead mode, the higher the performance benefit of runahead execution. On the other hand, speculative code execution imposes overheads for saving and restoring state, and rolling the pipeline back to a proper state to resume normal operation after runahead execution. The lower the performance penalty of these overheads, the higher the performance gain. Consequently, maximizing the performance benefits of runahead execution requires (1) maximizing the number of useful prefetches per runahead interval, and (2) limiting the switching overhead between runahead mode and normal execution. We find that prior attempts to optimize the performance of runahead execution have shortcomings that impede them from adequately addressing both issues, leaving significant room for improvement.

Prior runahead proposals incur a high performance penalty due to speculative execution in runahead mode [24, 40, 42]. All the instructions beyond the stalling load are pseudo-retired and leave the reorder buffer (ROB) and processor pipeline, and the corresponding physical register file entries are freed—in other words, processor state is released. These instructions must be re-fetched and re-executed in normal mode. For memory-intensive applications—the main beneficiaries of runahead execution—the probability for blocking the ROB and invoking runahead execution is high and so is its incurred performance penalty for releasing processor state with every runahead invocation. To mitigate this overhead, prior work [42] engages runahead execution only for relatively long runahead intervals. By doing so, the benefits of prefetching in runahead mode outweigh its overheads. Unfortunately, this optimization does not reduce the severe penalty incurred whenever runahead execution is invoked. Moreover, it limits the opportunities to engage runahead execution, thereby degrading prefetch coverage.

The original runahead proposal [40, 42] limits the prefetch coverage that can be achieved. In runahead mode, the processor executes all the instructions it encounters to generate useful prefetch requests. However, not all these instructions are necessary to calculate load addresses and generate prefetch requests. Instructions that do not lead to long-latency loads waste execution cycles and occupy processor resources that could otherwise be used to generate useful prefetch requests. Hashemi et al. [24] filter out unnecessary instructions by storing the chain of instructions that generate the blocking load in a so-called runahead buffer. In runahead mode, the processor keeps replaying only this instruction chain from the runahead buffer in a loop, which enables turning off the processor front-end in runahead mode to save energy. However, similar to other runahead techniques, runahead buffer incurs the high performance overheads associated with invoking runahead mode. More importantly, runahead buffer limits

‡This work was done while visiting Ghent University.

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00040

prefetch coverage to only a single chain of instructions per runahead interval, while several benchmarks access memory through multiple chains. Limited prefetch coverage reduces the potential performance gains from runahead buffer.

In this paper, we propose *precise runahead execution (PRE)*, a technique that remedies the aforementioned shortcomings of prior runahead proposals [45]. PRE builds upon the key observation that the processor has sufficient unused resources in the issue queue and physical register file to continue speculatively executing instructions in runahead mode, eliminating the need to release processor state in the reorder buffer (ROB), issue queue (IQ), and physical register file (PRF). PRE uses *runahead register reclamation (RRR)*, a novel mechanism to manage free physical registers in runahead mode while preserving dependencies among instructions. Moreover, PRE stores all instructions in the backward slice of a long-latency load in a dedicated cache, called the *stalling slice table (SST)*. First, the PC of the stalling long-latency load is stored in the SST, then with every loop iteration, the register renaming unit is leveraged to recursively identify all instructions in the load's backward slice, which are then stored in the SST. In runahead mode, PRE receives decoded instructions from the front-end but executes only the ones that hit in the SST. Because PRE stores all long-latency load chains in the SST, it does not limit prefetch coverage to a single load chain.

PRE can be augmented with an additional buffer to store all the decoded instructions in runahead mode. When normal execution resumes, instructions are then dispatched from this buffer. Therefore, it is not necessary to fetch and decode runahead-mode instructions again. We leverage and extend the micro-op queue, typically present in modern-day processors to hold decoded micro-ops, to buffer micro-ops during runahead mode.

In summary, PRE's key contributions are:

- PRE *only* speculatively pre-executes slices of load instructions that lead to full-window stalls.
- PRE does not release processor state when entering runahead mode.
- PRE leverages the available issue queue and physical register file entries to speculatively execute instructions in runahead mode.
- PRE includes runahead register reclamation (RRR), a novel mechanism to quickly free physical registers in runahead mode.
- PRE optionally buffers decoded micro-ops during runahead mode in an extended micro-op queue to avoid re-fetching and re-decoding instructions, thereby saving energy.

Compared to an out-of-order core, the performance improvements achieved through runahead execution [42], runahead buffer [24], hybrid runahead (combining the best of runahead execution and runahead buffer), and PRE for a set of memory-intensive SPEC CPU benchmarks amount to 16%, 13.3%, 20%, and 38.2% on average, respectively. While hybrid runahead is energy-neutral relative to an out-of-order core, PRE reduces energy consumption by 6.8%.

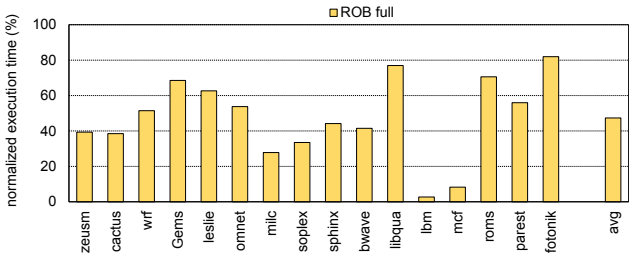


Fig. 1: Fraction of the execution time the ROB is full for memory-intensive benchmarks. *An out-of-order processor stalls on a full ROB for about half the time.*

II. Background and Motivation

In this section, we describe the original runahead proposal and the optimizations introduced in follow-on work. We then describe the shortcomings of prior runahead techniques.

A. Full-Window Stalls

In an out-of-order core, a load instruction that misses in the last-level cache (LLC) typically takes a couple hundred cycles to bring data from off-chip memory. Soon, the load instruction blocks commit and the core cannot make any progress. Meanwhile, the front-end continues to dispatch new instructions into the back-end. Once the ROB[1] fills up, the front-end can no longer dispatch instructions, leading to a *full-window stall*. Figure 1 shows that an out-of-order processor executing a set of memory-intensive SPEC CPU benchmarks spends about half of its execution time waiting for long-latency loads blocking the ROB (see Section IV for details about our experimental setup). We refer to the load instruction that causes a full-window stall as a *stalling load*, and to the backward chain of instructions that leads to a stalling load as a *stalling slice*.

B. Runahead Execution

Runahead execution [40] pre-executes an application's own code to prefetch data into the on-chip caches. Upon a full-window stall, the processor checkpoints the Program Counter (PC), architectural register file (ARF), the branch history register, and the return address stack (RAS). The processor enters runahead mode and marks the stalling load and its dependents as invalid. The processor pseudo-retires instructions without updating the processor architectural state to keep the execution moving forward speculatively. Once the stalling load returns, the pipeline is flushed and the checkpointed architecture state is restored. This marks the exit from runahead mode. The processor then fetches and executes instructions from the stalling load again.

Runahead execution incurs a significant performance and energy overhead by flushing and refilling the pipeline when returning to normal execution mode. Mutlu et al. [42] propose enhancements to the original runahead proposal to alleviate the impact of this high overhead. Mainly, they propose invoking runahead execution only when the runahead interval is long

[1]ROB and (instruction) window are used interchangeably.

enough to achieve high performance benefits that overshadow the overheads of runahead execution. In particular, they propose a policy to invoke runahead execution only if the stalling load was issued to memory less than a threshold number of cycles ago. They also propose another enhancement that prevents triggering runahead execution if it overlaps with an earlier runahead interval.

C. Future Thread

Future thread [6] shares the same purpose as runahead execution, while relying on two hardware threads, each with a dynamically allocated number of physical registers. When the main thread exhausts its allocated physical registers due to a long-latency load, it stalls and the processor switches to a second hardware context (i.e., the future thread) in an attempt to prefetch future stalling loads. This technique requires hardware support for two hardware contexts. Further, it exposes less MLP than runahead because the future thread needs to share resources with the main thread, which limits how far the future thread can speculate.

D. Filtered Runahead Execution

Both the original runahead and the future-thread techniques execute all instructions coming from the processor front-end. However, many instructions are not necessary to calculate the memory addresses used in subsequent long-latency loads. Hashemi et al. [24] propose a technique to track and execute only the chain of instructions that leads to a long-latency load. Upon a full-window stall, a backward data-flow walk in the ROB and store queue is performed to find a dependency chain that leads to another instance of the same stalling load. This chain is stored in a buffer called the *runahead buffer* that is placed before the rename stage. In runahead mode, the instruction chain stored in the runahead buffer is renamed, dispatched, and executed in a loop, instead of fetching new instructions via the front-end. Therefore, the front-end can be clock-gated to save dynamic power consumption in runahead mode. By executing only the stalling slice, this technique exposes more MLP per runahead interval than traditional runahead.

E. Shortcomings of Prior Techniques

Both traditional runahead execution and runahead buffer significantly improve single-threaded performance. However, their full potential is limited by the following key factors.

Flushing and Refilling the Pipeline. Runahead execution speculatively executes and pseudo-retires instructions. At the exit of runahead execution, the processor flushes the pipeline and starts fetching instructions from the stalling load. Performing this operation for every runahead invocation incurs significant performance and energy overheads. Assuming that the ARF can be saved/restored in zero cycles, we estimate that every runahead invocation incurs a performance penalty of approximately 56 cycles for a 192-entry ROB: (1) refilling the front-end (8 cycles, assuming an 8-stage front-end pipeline), plus (2) refilling the ROB by re-dispatching 192 instructions

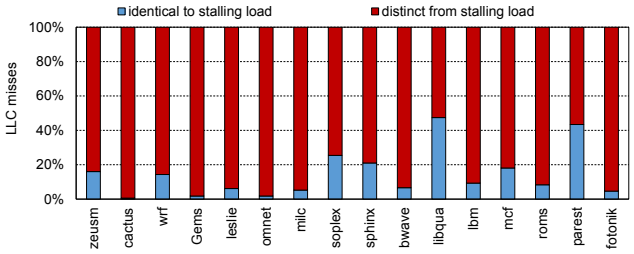


Fig. 2: Percentage of long-latency load misses during runhead that are identical to, versus distinct from, the stalling load. *Most of the long-latency loads during runahead mode differ from the stalling load.*

with a dispatch width of 4, starting from the stalling load (48 cycles). These cycles cannot be hidden and thus directly contribute to the total execution time. Our experimental results reveal that compared to an out-of-order core, traditional runahead execution improves performance by 16% on average. However, if the instructions that occupy the ROB when the core enters runahead mode would not need to be re-fetched and re-processed after exiting runahead mode, the speedup has the potential to reach 22.8%.

Limited Prefetch Coverage. Traditional runahead execution has limited prefetch coverage because it executes all future instructions in runahead mode, which limits how deep in the dynamic instruction stream runahead execution can speculate. Runahead buffer filters and executes only the most dominant stalling slice per runahead interval. Runahead buffer assumes that the load that triggers runahead execution is likely to recur more than any other load within the same runahead interval. Therefore, it decides to replay only the chain of instructions that produces future instances of the same stalling load. Although runahead buffer enables runahead execution to speculate further down the instruction stream, it is limited to a single slice. Unfortunately, this does not match the characteristics of applications that access memory through a diverse set of instruction slices and multiple different load instructions.

Figure 2 classifies the long-latency loads (i.e., loads that miss in the last-level cache) that are encountered in a runahead interval into either identical to, or distinct from, the stalling load that initiated the runahead interval. The figure shows that most of the long-latency loads that are encountered in a runahead interval differ from the stalling load that triggered runahead execution. Relying on a single dominant stalling load per interval, as in runahead buffer, therefore neglects major prefetching opportunities. (Note further that miss-dependent misses that appear in the dependence chain determined by runahead buffer cannot be prefetched—miss-dependent misses require a prediction mechanism such as address-value delta [43] or require migrating the dependency chain to the memory controller [25].)

In general, we find that memory-intensive applications access off-chip memory through multiple load slices. Figure 3 categorizes all runahead intervals according to the number of unique long-latency loads each interval contains. Most of the runahead intervals feature off-chip memory accesses via

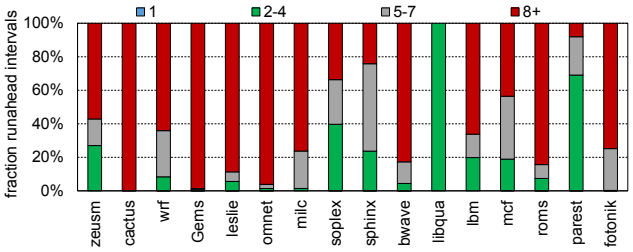


Fig. 3: Runahead intervals categorized by the number of unique long-latency loads. *Most runahead intervals feature multiple unique long-latency load instructions.*

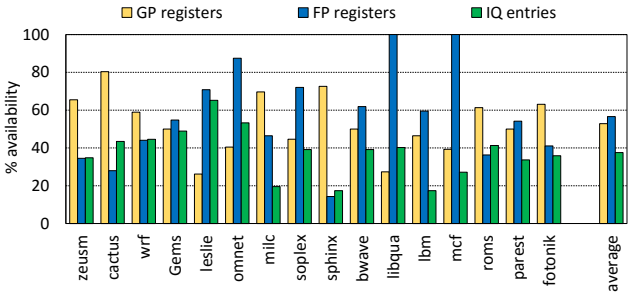


Fig. 4: Percentage general-purpose (GP) registers, floating-point (FP) registers and issue queue (IQ) entries that are available upon a full-window stall due to a long-latency load blocking commit. *About half the issue queue and physical register file entries are available upon a full-window stall.*

multiple unique load instructions.

Short Runahead Intervals. The proposed enhancements to runahead execution prevent initiating runahead mode if the runahead interval is estimated to be short. For such cases, the overhead of invoking runahead execution outweighs its benefit [42]. However, a significant fraction of runahead intervals are short. We find that 40% of the runahead intervals are shorter than 56 cycles for the memory-intensive workloads—recall 56 cycles is the overhead for refilling the pipeline after a runahead interval as previously determined. Excluding short runahead intervals thus limits how often runahead is triggered, which wastes significant opportunity to enhance MLP.

III. Precise Runahead Execution

In this work, we propose *precise runahead execution (PRE)* to alleviate the limitations of prior runahead proposals. PRE improves prefetch coverage over prior proposals by prefetching all stalling slices in runahead mode—unlike runahead buffer—and executing only the instruction chains leading to the loads—unlike the original runahead proposal. Moreover, PRE does not release processor state when entering runahead mode, hence it does not need to flush and refill the pipeline when resuming normal mode. This reduces the cost for invoking runahead execution.

We first describe the key insights that inspire the design of PRE, after which we describe PRE's architecture and operation in detail.

A. PRE: Key Insights

PRE builds on three key insights.

Insight #1: There are enough available physical register file (PRF) and issue queue (IQ) resources to initiate runahead execution upon a full-window stall. To execute an instruction, the processor minimally needs a physical register to hold the instruction's destination value plus an issue queue entry for the instruction to wait until an execution unit becomes available. Figure 4 shows the percentage of available (i.e., unused) processor issue queue and physical register file entries at the entry of runahead mode. On average, 37% of the issue queue entries, 51% of the integer registers and 59% of the floating-point registers are free. This is not an artifact of an unbalanced processor configuration. In fact, Section IV provides quantitative evidence that our baseline configuration is indeed a balanced design. We thus conclude that there are enough issue queue entries and registers upon a full-window stall to initiate the speculative execution of instructions that lead to anticipated future long-latency load misses.

Insight #2: There is no need to pre-execute all instructions during runahead mode. Instead we can speculate deeper in the dynamic instruction stream by only pre-executing stalling load slices. The majority of instructions executed during runahead execution occupy core resources (e.g., PRF, IQ, ALU) without actually contributing to generate useful prefetches. Ideally, we only need to speculatively execute instructions that lead to future long-latency load stalls, i.e., we need to execute the producers of the long-latency loads and not their consumers. This not only reduces the core resources needed during runahead execution, it also allows for speculating deeper down the dynamic instruction stream and extract more useful prefetches. PRE achieves this by identifying and speculatively executing *stalling load slices*, i.e., backward slices of long-latency loads that lead to full-ROB stalls.

Insight #3: IQ resources are quickly recycled during runahead execution. Recycling PRF resources requires a novel mechanism that is different from conventional register renaming schemes. Stalling load slices are relatively short chains of dependent instructions. These chains of load-producing instructions occupy IQ resources for only a short time, i.e., instructions wait for their input operands for a few cycles and then execute. In contrast, the load consumers hold on to IQ resources as they wait for the load values to return from memory. In other words, PRE is able to quickly recycle IQ resources by only executing stalling load slices during runahead mode. The situation is different for the physical register file: stalling load slices hold up PRF resources if they are released using conventional register renaming. PRE therefore includes a novel register reclamation mechanism to quickly recycle physical registers in runahead mode.

Figure 5 depicts a schematic diagram of an out-of-order core supporting PRE. The following subsections describe its operation in detail.

B. Entering Precise Runahead Execution

As in prior techniques, PRE is invoked on a full-window stall. PRE enters runahead mode after checkpointing the PC of the instruction past the full-ROB, the register alias table

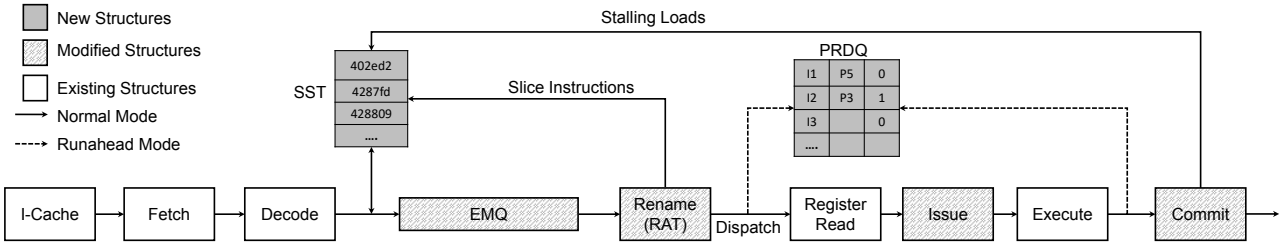


Fig. 5: Core microarchitecture to support precise runahead execution.

(RAT), and the RAS. The instructions filling the ROB can still execute as they do in normal mode. However, no instructions are committed from the ROB in runahead mode. Therefore, no updates are propagated to the ARF and the L1 D-cache. During runahead execution, PRE dynamically identifies the instructions that are part of potential stalling slices as they arrive from the decode unit (as described in the next section), and the core speculatively executes them.

C. Identifying Stalling Slices

PRE tracks the individual instructions that form a stalling slice in a new cache that we call the *stalling slice table (SST)*. As Figure 5 shows, the SST is accessed after the decode stage. The SST is a fully-associative cache that contains only instruction addresses (i.e., PCs). If an instruction address hits in the SST, that instruction is part of a stalling slice. Whenever a stalling load blocks the ROB, we store it in the SST. To facilitate tracking the chain of instructions that leads to that load, we extend each entry in the RAT to hold the PC of the instruction that last produced that register. When the register renaming unit maps the destination architectural register of an instruction to a new physical register, it also updates the RAT entry corresponding to that architectural register with the PC of the instruction.

We track the stalling slices in an iterative manner. First, the stalling load is stored in the SST. When the stalling load is decoded again, e.g., in the next iteration of a loop, the PC of the stalling load hits in the SST. PRE checks the RAT entry for the load's source registers to find the PCs of the instructions that last produced those registers; these PCs are then stored in the SST. Similarly, whenever an instruction hits in the SST in the following iterations, we track the PC information of its producer instructions and add those to the SST as well. This iterative process effectively builds up the stalling slice in the SST. PRE follows this same process for all stalling loads. By tracking all stalling slices in the SST, PRE does not limit prefetch coverage to a single slice as in the runahead buffer proposal.

Branch instructions are not part of a stalling load slice because they are not involved in the load address calculation. Therefore, branch instructions are not stored in the SST. A branch instruction can modify the stalling slice by changing the producer of one instruction in the slice, potentially forming two slices that lead to the same load instruction. PRE simply identifies the new producers and adds them to the SST. In the following iterations, PRE builds the whole slice in the SST similar to any other slice. In the end, SST tracks all slices that lead to stalling loads.

We find that an SST of limited size is effective at capturing stalling slices to generate useful prefetches in a runahead interval. As the application progresses to a new loop, new stalling slices are identified and stored in the SST. Old and unused stalling slices are automatically evicted from the SST. It may happen that a slice is not complete in the SST, e.g., while being constructed, however, the slice will soon be completed in the next few iterations. We find that an SST with 128 entries is sufficient to gain the majority of the performance benefits of runahead execution (see Section III-H).

D. Execution in Runahead Mode

PRE filters and speculatively executes all stalling slices that follow the stalled window using the SST. After instruction decode, PRE executes only the instructions that hit in the SST because they are necessary to generate future loads. PRE achieves the benefits of filtered runahead execution as with runahead buffer because it executes only the stalling slices. However, because the SST stores all stalling slices, PRE manages to execute *all* potential stalling slices, which leads to much improved prefetch coverage.

Instructions issued in runahead mode use only the free registers that are unused when runahead mode is triggered. These registers are allocated and recycled in runahead mode without affecting the physical registers allocated in normal execution. PRE properly maintains dependencies among the executed instructions and manages the allocation and reclamation of registers in runahead mode as described in Section III-E. At the same time, the processor continues executing the non-speculative instructions that already occupy the ROB. The results are written to the physical destination registers that were allocated before triggering runahead execution. When the processor resumes normal operation, it restores the architectural state it checkpointed upon runahead entry. Only instructions that were fetched in runahead mode need to be fetched and processed again. The physical registers that were free prior to runahead execution are reclaimed. The physical registers that hold values written by instructions in the ROB in runahead mode can properly update the architectural state and get reclaimed when their respective instructions retire in normal mode.

In runahead mode, PRE executes all the slices generated by the front-end of the processor. The front-end relies on the branch predictor to steer the flow of execution in runahead mode. PRE does not update the state and history of the

Register renaming and its outcome

inst. id	instruction	dst	src1	src2	register to free
I1	add r1 ← r2, r3	P1	P2	P3	
I2	mul r2 ← r1, r4	P5	P1	P4	P2
I3	ld r1 ← mem[x]	P6			P1
I4	add r2 ← r1, r3	P7	P6	P3	P5
I5	add r2 ← r4, r5	P9	P4	P8	P7
I6	sub r1 ← r2, r6	P11	P9	P10	P6

PRDQ

inst. id	register to free	executed ?
I1		1
I2	P2	1
I3	P1	0
I4	P5	0
I5	P7	1
I6	P6	0

Fig. 6: Recycling physical registers during precise runahead execution.

branch predictor during runahead execution. However, branch instructions that reside in the ROB can be resolved in runahead mode and update the predictor as they would in normal mode. If a branch instruction in the ROB turns out to be mispredicted, the processor discards all wrong-path instructions (including runahead instructions, if any), flushes the pipeline, and resumes normal execution.

E. Runahead Register Reclamation

PRE requires sufficient issue queue entries and physical registers to run ahead. As reported in Section III-A, such resources are usually available when entering runahead mode. Stalling slices are usually short and therefore issue queue entries are quickly reclaimed and are unlikely to hinder forward progress of runahead execution. In all of our experiments, we did not observe issue queue pressure during runahead.

PRE requires special support for reclaiming physical registers during runahead execution. In an out-of-order core, a physical register can be freed only when the last consumer of the renamed architectural register commits [62]. Since instructions that are fetched in runahead mode are discarded after they finish execution, we cannot rely on the conventional renaming policy to free physical registers. Thus, we devise a new mechanism, called *runahead register reclamation (RRR)*, to free physical registers in runahead mode. RRR relies on a new FIFO hardware structure, called the *precise register deallocation queue (PRDQ)* in Figure 5.

Figure 6 illustrates RRR in more detail. Each entry in the PRDQ has three fields: an instruction identifier, a physical register (tag) to be freed, and an 'execute' bit that marks whether the instruction has completed execution. The figure also provides a code example to help explain the operation of the PRDQ. The instructions in the example are numbered following program order. For example, instruction `I2` precedes instruction `I4` in program order. The figure shows the instructions after the register renaming stage. In this code example, instruction `I4` reads the value of architectural register `r1` from physical register `P6`, which is written by instruction `I3`. `I4` also reads the value of architectural register `r3` from physical register `P3` written by an older instruction not shown in the code example.

PRDQ entries are allocated in program order at the PRDQ tail. Register renaming maps a free physical register to the destination architectural register of an instruction in runahead mode. We mark the old physical register mapped to the same (destination) architectural register in the PRDQ entry. A PRDQ entry is deallocated when the instruction is executed (i.e., 'execute' bit is set) and reaches the PRDQ head. PRDQ deallocation is also done in program order. The old physical register associated with the instruction is freed upon deallocation. For example, in Figure 6, the renaming unit maps the destination architectural register of instruction `I4` (i.e., `r2`) to physical register `P7` and marks the old physical register mapped to `r2` (i.e., `P5`) to be freed when `I4` is retired and deallocated from the PRDQ.

While instructions may execute and thus mark the 'execute' bit out-of-order, in-order PRDQ deallocation guarantees that a physical register is freed only when there are no more instructions in-flight that may possibly read that register. The PRDQ is only enabled in runahead mode and its entries are discarded once the processor returns to normal mode.

F. Exiting Precise Runahead Execution

The core exits runahead mode when the stalling load returns. Upon exit, the core resumes normal execution after having restored the checkpointed PC, RAT, and RAS. As instructions are preserved in the ROB, the core starts committing instructions starting from the stalling load. The front-end re-directs fetch from the first instruction after the full-window stall, i.e., the PC which was checkpointed when entering runahead mode.

G. Front-End Optimization

PRE executes future stalling slices for the entire length of a runahead interval. During this time, PRE requires the front-end of the processor to keep fetching and decoding instructions. Therefore, the front-end has to remain active during runahead mode. The instructions fetched in runahead mode are fetched and processed again for execution in normal mode. This increases the energy overhead in the processor front-end for PRE compared to runahead buffer [24].

To avoid wasting the work and energy of the front-end in runahead mode, we propose the extended micro-op queue (EMQ) as shown in Figure 5. Superscalar out-of-order processors typically feature a micro-op queue to hold micro-ops after the instruction decode pipeline stage. For example, Intel Skylake uses a micro-op queues of 64 entries [29]. We propose extending the number of entries of the processor's micro-op queue, hence the name EMQ. The micro-op queue is a circular FIFO buffer and is extended without significantly impacting the complexity of the design. We augment PRE with an EMQ to store the micro-ops generated in runahead mode.

When using the EMQ, PRE stores all the decoded instructions in runahead mode, including the ones that hit in the SST. When the processor resumes normal execution, it does not need to fetch and decode these instructions again. Note that with this optimization, the number of speculatively executed instructions in runahead mode is constrained by the size of the EMQ. When the EMQ fills up, the core stalls until the stalling load returns, at which point, the processor exits runahead mode. Alternatively, the processor can continue fetching instructions beyond the size of EMQ for the whole runahead interval. In this case, the processor only needs to re-fetch the instructions that could not be buffered in the EMQ during runahead execution. This design alternative, however, is similar to PRE's original

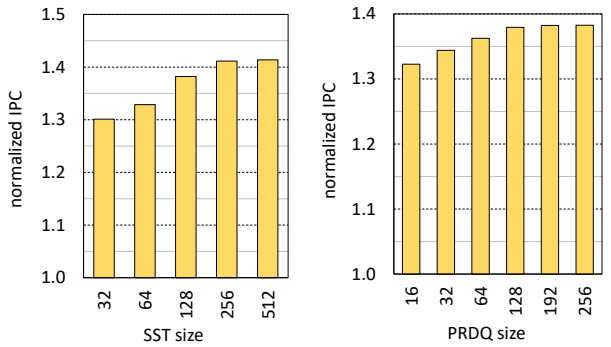


Fig. 7: Performance impact of changing the sizes of the SST and PRDQ. Performance is normalized to the OoO core. *An SST size of 128 entries balances performance and hardware cost; performance saturates for PRDQ size of 192 entries.*

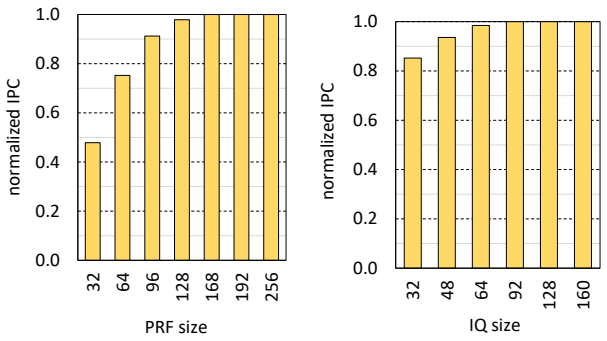


Fig. 8: Impact of PRF and IQ sizes on performance while keeping other configuration parameters constant. *Overall, the baseline OoO core with 168 PRF entries and 92 IQ entries is a balanced configuration.*

design and does not lead to significant variation in its energy and performance profile.

Engaging the EMQ is an optional design optimization. It is not mandatory for PRE's runahead operation. As we show in Section V-C, augmenting PRE with an EMQ of various sizes leads to different design points that trade off performance for energy. Designers can select a suitable design choice based on the available area and energy budgets, and performance goals.

H. Hardware Overhead

As mentioned before, PRE relies on the newly proposed SST and PRDQ. We conduct a sensitivity analysis to empirically select their sizes. Figure 7 reports the impact of varying the SST and PRDQ sizes on performance (normalized to a baseline OoO core). To balance hardware cost and performance, we opt for an SST with 128 entries; increasing the SST size beyond 128 entries leads to a minor gain in performance while incurring a significant hardware cost. We set the PRDQ size to 192 entries because it achieves the best performance and its hardware cost is small.

An SST with 128 entries each with a 4-byte tag requires 512 Bytes of storage. An entry in the PRDQ consists of a single bit to indicate that the instruction has finished execution, an 8-bit tag for the physical register to free, and 12 bits (assuming a maximum of 4096 runahead instructions) to give each instruction explored in runahead mode a unique ID. This adds up to a total of 504 Bytes. Additionally, we extend each mapping of the 64-entry RAT by 4 bytes for a total of 256 Bytes. This leads to a total hardware cost of 1.24 KB. When PRE is augmented with an (optional) EMQ, the hardware overhead is increased according to the selected EMQ size, with each EMQ entry requiring 4 Bytes to hold a micro-op. In comparison, runahead buffer incurs a hardware cost of about 1.7 KB and uses expensive CAM lookups in the ROB to determine stalling slices. Overall, the hardware cost and complexity of PRE is smaller compared to the runahead buffer proposal.

IV. Methodology

Simulation Setup. We evaluate precise runahead execution using the cycle-level, hardware-validated Sniper 6.0 [10] simulator, using its most accurate core model. The configuration for our baseline out-of-order core is provided in Table I. The sizes of the ROB, the physical register files, and the micro-op queue are based on the Haswell architecture [19, 23]; the size of the issue queue is set as in the runahead buffer paper [24] for fair comparison. We verify that this baseline configuration is indeed balanced, see Figure 8, i.e., the physical register file (PRF) and issue queue (IQ) sizes are the minimum sizes that lead to the best performance for the given ROB size. We assume that hardware prefetching is not enabled in our baseline core. However, we do evaluate the impact of hardware prefetching in Section V-D. We consider an 8 KB TAGE-SC-L branch predictor as implemented for the 2016 Branch Prediction Championship [55].

Frequency	2.66 GHz
Type	out-of-order
ROB size	192
Issue queue size	92
Load queue size	64
Store queue size	64
Micro-op queue size	28
Pipeline width	4
Pipeline depth	8 stages (front-end only)
Branch predictor	8 KB TAGE-SC-L
Functional units	3 int add (1 cyc), 1 int mult (3 cyc), 1 int div (18 cyc), 1 fp add (3 cyc), 1 fp mult (5 cyc), 1 fp div (6 cyc)
Register file	168 int (64 bit) 168 fp (128 bit)
SST size PRDQ size	128 entry, fully assoc, 6r 2w 192 entry, 4r 4w
L1 I-cache L1 D-cache Private L2 cache	32 KB, assoc 4, 2 cyc 32 KB, assoc 8, 4 cyc 256 KB, assoc 8, 8 cyc
Shared L3 cache	1 MB, assoc 16, lat 30 cyc
Memory	DDR3-1600, 800 MHz ranks: 4, banks: 32 page size: 4 KB, bus: 64 bits `tRP-tCL-tRCD`: 11-11-11

TABLE I: Baseline configuration for the out-of-order core.

Power. We use McPAT [37] to calculate power consumption assuming a 22 nm chip technology. We calculate power for the SST, EMQ and PRDQ using CACTI 6.5 [38] and add those

estimates to the McPAT power numbers. We report system power (processor plus main memory).

Cycle Time. We model the impact of the newly added hardware structures on processor cycle time using CACTI 6.5 [38]. We assume that the front-end can deliver up to six micro-ops per cycle to the micro-op queue. Therefore, the SST has 6/2 read/write ports. In runahead mode, we can check up to six micro-ops per cycle in the SST. PRDQ is an in-order queue with 4/4 read/write ports. The cycle time for accessing the SST and PRDQ equals 0.314 ns and 0.102 ns, respectively. Since this is below the processor cycle time (0.375 ns), we conclude that the accesses to the SST and PRDQ do not impact processor timing. (The SST can be pipelined, if needed, since it is not on the critical path.)

Workloads. We evaluate a total of 16 memory-intensive benchmarks from the SPEC CPU2006 and SPEC CPU2017 suites. From the CPU2006 suite, we select the same benchmarks as runahead buffer [24], and we maintain the same order when presenting our results. Compared to SPEC CPU2006, there are fewer memory-intensive benchmarks in the CPU2017 suite and, even though some benchmarks (e.g., `bwaves`) have multiple input data sets, their fraction of full-window stalls is similar in our setup. The three new memory-intensive benchmarks we have included from the SPEC CPU2017 suite are `roms_r_1`, `parest_r_1` and `fotonik3d_r_1`. We create 1 Billion instruction SimPoints [56] for each benchmark.

V. Evaluation

We compare the following four mechanisms:

- OoO: Our baseline out-of-order core from Table I.
- RA: Runahead execution, as explained in Section II-B, with the following enhancements [24, 42]:
  - There are no overlapping runahead intervals.
  - Runahead execution is triggered only when the stalling load instruction was issued to memory less than 250 cycles earlier.
- RA-buffer: The runahead buffer mechanism explained in Section II-D. In runahead mode, the front-end of the processor is clock-gated and the dominant stalling load slice for each runahead interval is executed from the runahead buffer. We assume all the chains are stored in a chain cache. Therefore, no extra overhead is required to perform backward walks in the ROB.
- RA-hybrid: The hybrid runahead approach selects the runahead technique (RA or RA-buffer) that yields the highest performance on a per-application basis.
- PRE: The precise runahead execution proposal as described in this paper.

We use *instructions per cycle (IPC)* to quantify performance. We calculate average performance across all benchmarks using the harmonic mean IPC across all benchmarks.

A. Performance

Figure 9 reports performance for the various runahead techniques, normalized to the baseline OoO core. While RA and RA-buffer improve performance over the OoO

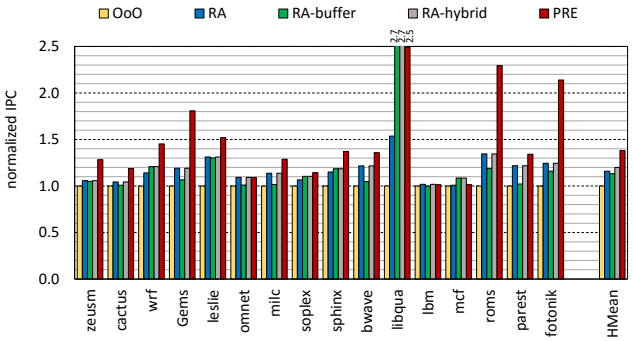


Fig. 9: Performance (IPC) normalized to an out-of-order core for runahead execution, runahead buffer and precise runahead execution. *PRE improves performance by 38% on average compared to the baseline out-of-order core.*

core by on average 16.0% and 13.3%, respectively, RA-hybrid which selects the best of both techniques improves performance by 20%. PRE on the other hand manages to improve performance by 38.2%. This is an additional 18.2% improvement over prior runahead techniques. Most of the applications gain a significant performance improvement with PRE. In general, we find that applications that spend more time waiting on a full-window stall have a higher chance to benefit from PRE. PRE achieves the highest performance improvements for `GemsFDTD`, `leslie3d`, `libquantum`, `roms` and `fotonik`. As Figure 1 shows, these applications spend more than 60% of their execution time on full-window stalls, providing PRE a significant opportunity to generate useful prefetches. The performance improvements for these applications range from 52% up to more than 2×, see `libquantum`, `roms` and `fotonik`. Other applications that spend less time waiting for long-latency loads like `zeusmp`, `wrf`, `milc`, `sphinx3`, `bwaves` and `parest` still achieve a significant performance improvement that ranges between 20% and 40%.

The significant performance improvement of PRE relative to prior runahead techniques comes from its higher prefetch coverage and the fact that it avoids flushing and re-filling the pipeline when leaving runahead mode. However, we find a few outlier cases where PRE has only a minor benefit compared to either the OoO or to prior runahead techniques. We observe that none of the runahead techniques significantly improve performance of the OoO core for `lbm`. This benchmark experiences full-window stalls for only 2.7% of the total execution time because the pipeline stalls on other resources. Therefore, the opportunity to prefetch in runahead mode is quite small. On the other hand, `omnetpp` is characterized by long stalling slices, as corroborated by [24]. The long stall slices limit PRE's opportunity to explore multiple slices per runahead interval. Therefore, PRE performs similarly to prior runahead execution for `omnetpp`.

The only benchmarks that benefit from RA-buffer more than PRE are `libquantum` and `mcf`. For `libquantum`, about 50% of the load instructions that access memory in a runahead interval are identical to the stalling load as Figure 2 shows. The rate at which RA-buffer executes the

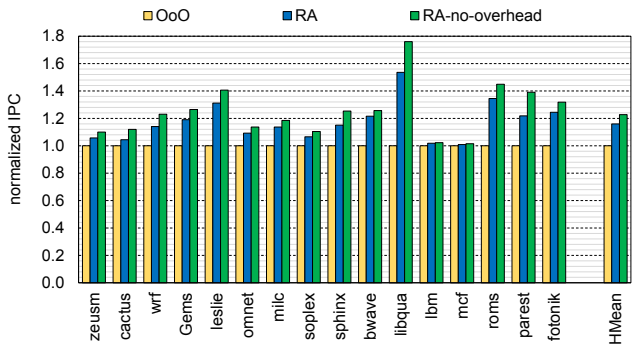


Fig. 10: Performance impact of flushing the pipeline when leaving runahead mode and refilling it when resuming normal execution in RA. *PRE avoids this overhead as it does not need to flush the pipeline when leaving runahead mode.*

same stalling slice to generate prefetches exceeds that of PRE, which has to dynamically determine the slices. The benefits of the faster prefetch generation in a limited runahead interval for `libquantum` outweigh the benefits of finding all slices. On the other hand, `mcf` is characterized by its high branch misprediction rate. This means that both PRE and prior runahead techniques invoke useless runahead intervals that execute wrong-path instructions, and thus do not improve performance. Branch instructions that wait for the stalling load to be resolved benefit from RA-buffer because it prefetches only stalling load slices. RA-buffer is particularly beneficial for load-dependent branches that are mispredicted. Therefore, it manages to slightly improve performance over PRE which dynamically explores all stall slices.

We now further analyze the sources of performance improvement for PRE over prior runahead techniques.

Pipeline Refill Overhead. PRE does not need to flush and refill the pipeline when resuming normal mode. This alone gives PRE a significant performance improvement over the original runahead proposal. Even with the enhancements introduced to the original runahead technique, the overhead of flushing the pipeline when leaving runahead mode and refilling it starting from the stalling load still limits its performance improvement. Figure 10 demonstrates the significant impact of flushing and refilling the processor pipeline on RA's performance improvement. Every exit from the runahead mode is followed by a pipeline bubble of at least 56 cycles—8 cycles to re-fill the front-end and 48 cycles to re-dispatch the same instructions to the ROB. As the figure shows, RA improves the performance of the OoO core by 16% on average. The performance improvement jumps to 22.8% when the flushing and refilling overhead is avoided.

MLP. PRE improves the degree of MLP that is exposed over prior proposals, for three reasons. First, PRE triggers runahead execution even for relatively short runahead intervals. This allows PRE to invoke runahead execution 1.8× more than RA and RA-buffer. Second, PRE executes only the stalling slices, which enables PRE to uncover long-latency loads at a higher rate than RA per runahead interval, and thus speculate deeper down the dynamic instruction stream. Third, PRE targets multiple stalling load slices during runahead execution in

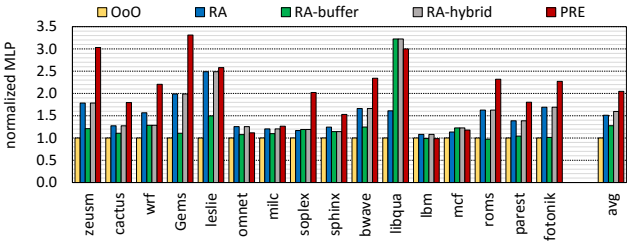


Fig. 11: Normalized MLP. *PRE improves MLP by 2× compared to an out-of-order core.*

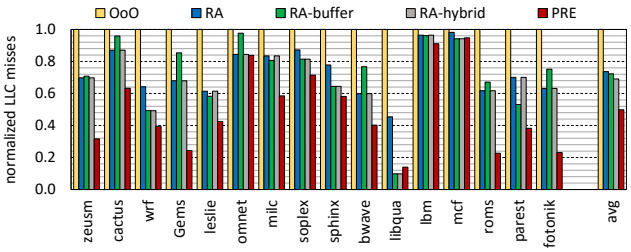


Fig. 12: Normalized LLC miss count during normal (non-runahead) execution. *PRE's accurate prefetches reduce the number of LLC misses by 50% compared to an OoO core.*

contrast to RA-buffer which speculatively executes only one stalling slice in a loop.

As Figure 11 shows, the MLP generated by RA, RA-buffer, RA-hybrid, and PRE is 1.5×, 1.3×, 1.6×, and 2× higher than for the OoO core. PRE improves MLP for most of the applications, except for the few outlier applications that were previously discussed. In general, the higher MLP of PRE reflects its superior prefetch quality, which leads to higher overall performance. It is worth noting that although RA-buffer can generate about 2× more memory requests than RA per runahead interval as reported in [24], overall performance is not proportionally improved.

LLC Miss Rate. Figure 12 reports normalized LLC miss rate in normal mode for all the runahead techniques. All runahead techniques reduce the number of LLC misses observed during normal mode. However, we find that PRE covers more LLC misses than any other prior runahead technique. On average, RA, RA-buffer, and RA-hybrid reduce the number of LLC misses by 26.4%, 27.7% and 31%, respectively, whereas PRE reduces the number of LLC misses by 50.2%. This higher reduction in LLC miss rate is a result of covering more stalling slices deeper down the dynamic instruction stream.

B. Energy Analysis

Figure 13 shows the energy consumption for all runahead techniques normalized to the OoO core. RA increases energy consumption of an OoO core by 2.4% on average. RA-buffer clock-gates the front-end during runahead mode to reduce energy overhead to only 0.4% relative to the baseline OoO core. RA-hybrid slightly reduces the energy consumption compared to RA-buffer. In general, we find that the significant performance improvement of PRE allows it to complete the same task with less energy than the other techniques for most

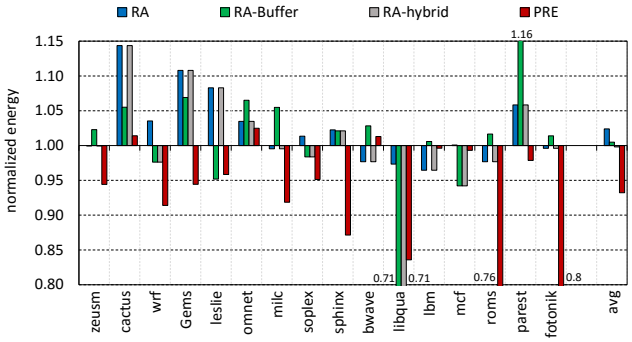


Fig. 13: Normalized energy consumption. *PRE reduces energy consumption by 6.8% compared to an out-of-order core, while runahead execution slightly increases energy consumption or is energy-neutral.*

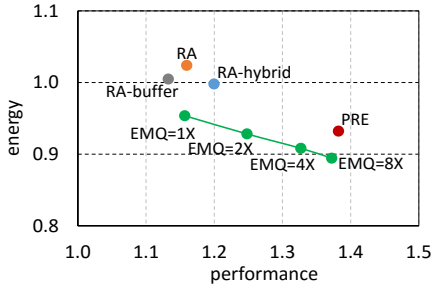


Fig. 14: Performance versus energy normalized to the OoO core. *PRE improves performance and reduces energy consumption compared to an out-of-order core. Increasing the size of the (optional) EMQ further reduces energy consumption and presents an energy-performance trade-off.*

of the applications. Similar to our earlier discussion, only few outlier cases such as `libquantum` and `mcf` consume less energy using RA-buffer than with PRE. On average, PRE performs the same task with 6.8% less energy compared to the baseline OoO core.

C. Front-End Energy Optimization

PRE requires the front-end of the processor to remain active in runahead mode to find stalling slices further down the dynamic instruction stream. Upon resuming normal mode, the processor fetches and executes all the instructions that were *fetched* in runahead mode again. In Section III-G, we proposed the EMQ as an optimization to save the energy consumed by the front-end in runahead mode. The EMQ is a design choice that trades off performance for energy.

Figure 14 shows the performance-energy trade-off for PRE with an EMQ of different sizes in multiples of the ROB size. (For example, an EMQ of size 2 × has 384 entries.) Without an EMQ, PRE keeps exploring the code throughout the entire runahead interval, leading to the highest performance improvement, however, this requires refetching instructions upon return to normal mode. With a limited EMQ, PRE can save the work of the front-end but may halt runahead execution before the end of the runahead interval. In contrast, larger EMQs enable PRE to explore more code than smaller ones, leading

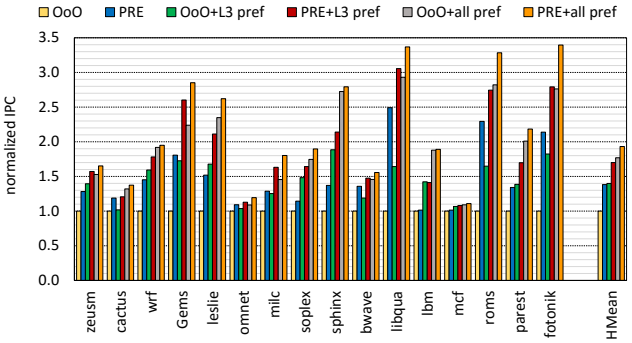


Fig. 15: Performance relative to the baseline OoO core (without prefetching) when hardware prefetching is enabled at the LLC and all the cache levels. *PRE improves performance even when conventional stride prefetching is enabled at the LLC and all the cache levels.*

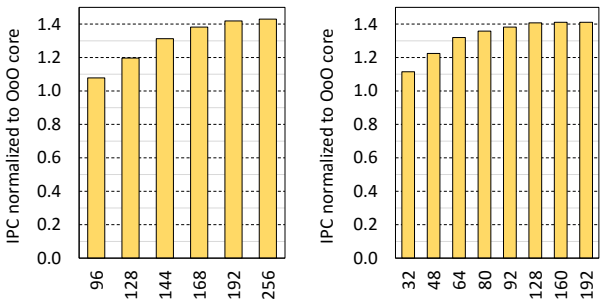
to higher performance and saving more work in the front-end. Thus, with a larger EMQ size, performance improves and energy consumption decreases. With a sufficiently large EMQ, it is possible to find design points that achieve comparable performance to PRE (without EMQ) while significantly saving energy, such as in the case for the EMQ=8× and EMQ=4× configurations. This comes at an increase in hardware cost though, e.g., an EMQ=4× storing 4 Bytes per entry requires 3 KB.

Interestingly, Figure 14 also shows that augmenting PRE with an EMQ provides better performance-energy trade-off points than prior runahead techniques even with limited EMQ sizes. For example, for the EMQ=1× configuration, PRE yields higher performance than RA-buffer at a lower energy cost. Similarly, for the EMQ=2× configuration, PRE yields higher performance than all prior runahead techniques at a lower energy cost. Whether to use an EMQ or not, and which EMQ size to select, are design alternatives that can be selected at design time based on the available energy and area budgets.

D. Architecture Sensitivity

Hardware Prefetching. Hardware prefetchers and runahead techniques both aim at bringing data into the on-chip caches before it is needed by the workload. Generally speaking, hardware prefetchers exploit memory access patterns to predict which data to prefetch. On the other hand, runahead techniques generate prefetch requests by pre-executing the code. Both techniques are complementary to each other. If the hardware prefetchers are able to predict LLC misses and convert them into hits, runahead execution is not triggered. Conversely, when runahead techniques are effective at prefetching data, hardware prefetchers are invoked fewer times.

Figure 15 shows the performance improvement of the baseline OoO core and PRE when augmented with hardware prefetchers. We evaluate two configurations: (i) a stride-based LLC hardware prefetcher with 16 streams, and (ii) a stride-based hardware prefetcher with 16 streams incorporated at all levels in the hierarchy. PRE leads to significant performance improvements even for processor configurations with conven-



(a) PRF size (b) IQ size

Fig. 16: Performance improvement through PRE as a function of PRF and IQ sizes. *PRE improves performance even for PRF and IQ sizes that are underprovisioned.*

tional hardware prefetchers. For the configuration with the LLC prefetcher as a baseline, PRE improves performance by 21.5%. For the configuration with prefetchers engaged at all cache levels, PRE improves performance by 9.1%. The performance benefit obtained through PRE is expected to reduce with more aggressive hardware prefetching. Nevertheless, we conclude that PRE offers non-trivial performance improvements even under (aggressive) hardware prefetching.

Physical Register File. PRE leverages available PRF entries to speculate beyond a full ROB. Figure 16(a) quantifies PRE's average performance improvement as we scale the number of PRF entries (PRF=*N* means *N* integer and *N* floating-point registers). PRE performance is (obviously) sensitive to the number of physical registers. Small PRF sizes exhaust the number of available PRF entries, preventing PRE from speculating beyond the full ROB. Our baseline configuration assumes a balanced PRF size of 168 entries. Smaller PRF sizes, even if this leads to an unbalanced baseline design, would still experience a non-trivial improvement through PRE: 19.7% average performance improvement for a PRF size of 128 and 31.2% for a PRF size of 144.

Issue Queue. Similarly, PRE leverages available issue queue (IQ) sizes to speculate beyond a full ROB. Figure 16(b) reports the average performance improvement achieved through PRE as a function of IQ size. Small IQ sizes limit the number of resources that PRE can use during runahead mode, which limits the performance improvement achieved by PRE. Our baseline assumes an IQ size of 92. Smaller IQ sizes still enable PRE to achieve substantial performance improvements: 31.9% for an IQ size of 64 and 35.8% for an IQ size of 80.

LLC Size and Skylake. Data footprints for memory-intensive applications go beyond the LLC. Even when quadrupling the LLC size to 4 MB, PRE still achieves twice the performance of the best performing prior work (15% improvement for prior work versus 31% for PRE relative to baseline). We observe similar performance results and double the performance gain over prior work for a Skylake-like architecture (224-entry ROB, 97-entry IQ, and 180 physical registers). The average performance gains over the baseline (Skylake) core for RA, RA-buffer, RA-hybrid, and PRE amount to 13.1%, 12%, 17.2%, and 31.5%, respectively.

VI. Related Work

A large body of processor microarchitecture research has focused on improving single-thread performance over the past four decades. Various proposals scaled microarchitecture structures for better performance and energy-efficiency. Examples include proposals that dynamically scale operating voltage and clock frequency [8, 30, 50] or resize critical structures like issue queue [9, 20, 28, 34, 49] and caches [1, 2, 5] or throttle the front-end pipeline [39]. PRE fits in the category of work that performs some form of runahead execution, pre-execution or prefetching.

Runahead. PRE improves upon the runahead execution proposed within a single core [24, 40, 41, 42, 43, 44]. Since traditional runahead execution cannot prefetch dependent long-latency load instructions, address-value delta [43] predicts the data value of earlier long-latency load instructions to enable the execution of future ones. An enhanced memory controller [25] filters this chain of dependent long-latency load instructions and executes it at the memory controller; now, the dependent load instructions can execute as soon as the data is available from DRAM. Because the effective runahead interval shortens with the increasing size of the ROB, continuous runahead [26] proposes a tiny accelerator that is located at the last-level cache controller of a multi-core chip. The accelerator executes the dependency chain that leads to the highest number of full-window stalls within the core. However, the area overhead of the accelerator is 2% of a quad-core chip, and likely higher for a single core. Prior work has also proposed runahead threads in an SMT processor [51, 52, 66]. PRE is a runahead technique that does not require a separate core or runahead thread to pre-execute stalling slices.

Pre-Execution. This category of work executes performance critical instruction slices early in a software-only, hardware-only or a hardware-software cooperative fashion. Helper threads [32] and speculative precomputation [14] are software-only techniques that require a hardware context for early execution. Hardware-only techniques filter critical instruction slices from the back-end of a processor for early execution on a separate hardware context [15, 72]. Waiting instruction buffer (WIB) [36] and continual flow pipelines (CFP) [60] execute a large number of independent instructions by releasing the resources occupied by miss-dependent instructions. BOLT [27] builds upon CFP but reuses SMT hardware to rename deferred slices and introduces a set of mechanisms to avoid useless pre-execution of slices. Slipstream processors [61] improve performance and reliability by precomputing demand misses. Dependence graph precomputation (DGP) [3] dynamically precomputes and executes instructions responsible for memory accesses on a separate execution engine. Dual-core [70] and explicitly-decoupled architecture (EDA) [21, 22, 33, 48] use two hardware threads where one thread feeds its output to the other. Hardware-software cooperative techniques involve new instructions, advanced profiling, or binary translation for separating critical instruction slices, see for example DAE [57], speculative slice execution [71], flea-flicker multi-pass pipelining [7], braid processing [65], and OUTRIDER [16]. Instruction slices have also been exploited to improve the energy-efficiency

of both in-order and OoO processors [11, 35, 54, 63, 64]. PRE does not require a helper thread, hardware context, or support from software for converting demand misses into hits.

Prefetching. Hardware prefetchers are typically employed in modern processors [29]. Stride or stream prefetchers are able to prefetch common stride data access patterns that are independent of other memory accesses [17, 31, 47]. The accesses are either contiguous or separated by a constant stride. Address-correlating prefetchers require larger tables and target pointer-chasing access patterns [4, 12, 13, 58, 59, 67, 68, 69]. These prefetchers build on the premise that data structures are typically accessed in the same manner, generating the same cache misses repeatedly. Global history buffer (GHB) [46] splits the correlation table into two separate structures and also lowers the hardware overhead. PRE is implemented completely within the core and is orthogonal to other hardware prefetching techniques.

VII. Conclusion and Future Work

Runahead execution improves processor performance by accurately prefetching long-latency memory accesses. We show that the performance of prior runahead proposals is limited by the high overhead they incur and the limited prefetch coverage they achieve. Prior proposals release processor state when entering runahead mode and need to re-fill the pipeline when resuming normal operation. This operation introduces significant performance overhead. Moreover, prior proposals have limited prefetch coverage due to executing instructions that are unnecessary to generate prefetches as in the original runahead proposal, or due to not exploring all possible stalling loads as in runahead buffer.

In this paper, we propose *precise runahead execution (PRE)*, to alleviate the shortcomings of prior runahead proposals. We observe that at the entry of runahead mode, there are sufficient free PRF and IQ resources to speculatively execute instructions without having to release processor state. PRE does not incur the performance overhead of refilling the pipeline when resuming normal operation, by featuring a novel mechanism to quickly recycle physical registers in runahead mode. Furthermore, PRE tracks all stalling slices in a dedicated cache, which it executes in runahead mode, i.e., PRE filters unnecessary instructions and pre-executes all stalling slices to improve prefetch coverage. Our experimental evaluation shows that PRE outperforms recent runahead proposals by 18.2% on average, while reducing energy consumption by 6.8%.

For all runahead techniques including PRE, there is a risk of leaking information as instructions are executed speculatively. For mitigating such risk, we can exploit recently proposed techniques such as CleanupSpec [53] which adds small area and performance overhead for undoing the changes made to the cache hierarchy by speculative instructions. Investigating the interplay between (precise) runahead and recently proposed security mitigation techniques is subject of future work.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported by European Research Council (ERC) Advanced Grant agreement no. 741097, and FWO projects G.0434.16N and G.0144.17N. Josué Feliu was supported through a postdoctoral fellowship by the Generalitat Valenciana (APOSTD/2017/052).

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 248–259, Nov. 1999.

[2] D. H. Albonesi, R. Balasubramonian, S. G. Dropsbo, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12): 49–58, 2003.

[3] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–61, 2001.

[4] J. Baier and G. R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, SE-2(1): 54–62, 1976.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO)*, pages 245–257, 2000.

[6] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pages 26–37, 2001.

[7] R. D. Barnes, S. Ryoo, and W. W. Hwu. "Flea-flicker" multipass pipelining: an alternative to the high-power out-of-order offense. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–330, 2005.

[8] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 9–14, June 2000.

[9] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 147–156, 2003.

[10] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28, 2014.

[11] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. The load slice core microarchitecture. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 272–284, 2015.

[12] M. J. Charney. *Correlation-based Hardware Prefetching*. PhD thesis, 1995.

[13] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–209, 2002.

[14] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 14–25, July 2001.

[15] J. D. Collins, D. M. Tullsen, , and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO)*, pages 306–317, 2001.

[16] N. C. Crago and S. J. Patel. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 117–128, 2011.

[17] F. Dahlgren and P. Stenström. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 68–, 1995.

[18] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 68–75, July 1997.

[19] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf.

[20] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 230–239, 2001.

[21] A. Garg and M. C. Huang. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st International Symposium on Microarchitecture (MICRO)*, pages 306–317, 2008.

[22] A. Garg, R. Parihar, and M. C. Huang. Speculative parallelization in decoupled look-ahead. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 413–423, 2011.

[23] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation Intel Core processor. *IEEE Micro*, pages 6–20, 2014.

[24] M. Hashemi and Y. N. Patt. Filtered runahead execution with a runahead buffer. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 358–369, 2015.

[25] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Accelerating dependent cache misses with an enhanced memory controller. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 444–455, 2016.

[26] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[27] A. Hilton and A. Roth. BOLT: Energy-efficient out-of-order latency-tolerant execution. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[28] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium on Low Power Electronic Design (ISLPED)*, pages 32–37, 2004.

[29] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, Apr. 2019.

[30] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 347–358, Dec. 2006.

[31] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 364–373, June 1990.

[32] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computer Systems (TOCS)*, 22(3):326–379, 2004.

[33] S. Kondguli and M. Huang. R3-DLA (reduce, reuse, recycle): A more efficient approach to decoupled look-ahead architectures. In *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 533–544, 2019.

[34] Y. Kora, K. Yamaguchi, and H. Ando. MLP-aware dynamic instruction window resizing for adaptively exploiting both ILP and MLP. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, pages 37–48, 2013.

[35] R. Kumar, M. Alipour, and D. Black-Schaffer. Freeway: Maximizing mlp for slice-out-of-order execution. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–569, 2019.

[36] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 59–70, 2002.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec. 2009.

[38] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 694–701, 2011.

[39] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, (ISCA)*, pages 132–141, 1998.

[40] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, Feb. 2003.

[41] O. Mutlu, , J. Stark, and Y. N. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. *IEEE Computer Architecture Letters*, 4(1):2–2, 2005.

[42] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 370–381, June 2005.

[43] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (AVD) prediction: increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 233–244, 2005.

[44] O. Mutlu, H. Kim, and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26(1):10–20, 2006.

[45] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. *IEEE Computer Architecture Letters*, 18(1):71–74, 2019.

[46] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 96–105, 2004.

[47] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 24–33, 1994.

[48] R. Parihar and M. C. Huang. Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 662–677, 2014.

[49] D. V. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, and P. M. Kogge. Energy-efficient issue queue design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(5):789–800, 2003.

[50] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 560–563, Nov. 2001.

[51] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead threads to improve SMT performance. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 149–158, Feb. 2008.

[52] T. Ramirez, A. Pajuelo, O. J. Santana, O. Mutlu, and M. Valero. Efficient runahead threads. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 443–452, 2010.

[53] G. Saileshwar and M. K. Qureshi. CleanupSpec: An "undo" approach to safe speculation. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*, pages 73–86, 2019.

[54] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud. Long term parking (LTP): Criticality-aware resource allocation in OOO processors. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 334–346, 2015.

[55] A. Seznec. TAGE-SC-L branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[56] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, 2002.

[57] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–119, 1982.

[58] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, pages 252–263, 2006.

[59] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 69–80, 2009.

[60] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 107–119, Oct. 2004.

[61] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 257–268, 2000.

[62] H. Tabani, J. Arnau, J. Tubella, and A. Gonzalez. A novel register renaming technique for out-of-order processors. In *International*

Symposium on High Performance Computer Architecture (HPCA), pages 259–270, 2018.

[63] K. A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. Clairvoyance: Look-ahead compile-time scheduling. In *Proceedings of the International Conference on Code Generation and Optimization (CGO)*, pages 171–184, 2017.

[64] K. A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras. SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 328–343, 2018.

[65] F. Tseng and Y. N. Patt. Achieving out-of-order performance with almost in-order complexity. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 3–12, 2008.

[66] K. Van Craeynest, S. Eyerman, and L. Eeckhout. MLP-aware runahead threads in a simultaneous multithreading processor. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 110–124, 2009.

[67] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 222–233, 2005.

[68] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 79–90, 2009.

[69] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. Imp: Indirect memory prefetcher. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, pages 178–190, 2015.

[70] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 231–242, 2005.

[71] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, July 2001.

[72] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 172–181, June 2000.